Programming Languages for Computer Music Synthesis, Performance, and Composition

GARETH LOY

Computer Audio Research Laboratory, Center for Music Experiment and Related Research, University of California, San Diego, California 92093

CURTIS ABBOTT

Lucasfilm Ltd., P.O. Box 2009, San Rafael, California 94912

The development of formal, descriptive, and procedural notations has become a practical concern within the field of music now that computers are being applied to musical tasks. Music combines the real-time demands of performance with the intellectual demands of highly developed symbolic systems that are quite different from natural language. The richness and variety of these demands makes the programming language paradigm a natural one in the musical application of computers. This paradigm provides musicians with a fresh perspective on their work. At the same time, music is a very advanced form of human endeavor, making computer music applications a worthy challenge for computer scientists. In this paper we outline the traditional tasks and forms of representation in music, then proceed with a survey of languages that deal with music programming.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time systems*; D.3.2 [**Programming Languages**]: Language Classifications—*data-flow languages*; *extensible languages*; *nonprocedural languages*; *very high-level languages*; D.3.3 [**Programming Languages**]: Language Constructs—*coroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *interpreters*; *preprocessors*; E.1 [**Data**]: Data Structures; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods; J.5 [**Computer Applications**]: Arts and Humanities—*music*; K.2 [**Computing Milieux**]: History of Computing

General Terms: Languages

Additional Key Words and Phrases: Computer music, digital sound synthesis, music programming language design, music representation, real-time control languages

INTRODUCTION

With the advent of digital computer systems, advances in the traditional practices of music have been made possible. The ways in which computers can be employed in the service of music include signal processing, score representation, compositional assistance, and real-time control of the complex processes that go into creating, performing, and analyzing music. Solving musical problems with digital computers involves a broad class of issues in computer science, including questions of efficiency, adequacy of representation, and modeling. Because of this, researchers in the field of computer music have been required to keep abreast of and utilize current developments

Curtis Abbott's present address: Xerox Palo Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

CONTENTS

in computer science. Complimentarily, some computer scientists have discovered that music provides a fertile field for the study of issues in computer science [Minsky 1981; Roads 1980a; Smoliar 1971]. Some exciting possibilities for both music and computer science have begun to emerge from these two developments.

Certainly, the language used to describe music, and the theory implied in that language, is a baseline for any application of computers to music. Formal tools must be developed which express the desired phenomenon. For music, this is no easy task. Historically, the Western musical tradition has developed what we now refer to as *common practice notation* (CPN) to provide a static representation of musical compositions. What is denoted by CPN, however, is but the tip of a much greater body of oral knowledge and tradition in the practices of composition, performance, and analysis. The extent of this unwritten body of musical knowledge is made painfully evident by the attempt to develop formal tools to express it. And yet this is an initial requirement of anyone wishing to make or study music with computers. This has led to some of the developments in language and theory described in this paper.

Not only are computer tools being developed to represent various aspects of music, but the resulting representations are beginning to reflect changes in the nature of our understanding of music. These changes can be loosely characterized as a *generalization* of the degrees of freedom available to composers, musicians, and theorists in their approach to music. This advance is the direct result of a musician's willingness to be influenced by the methods and understanding of computer science and other disciplines, such as physics and psychoacoustics.

The effect on musical thinking is broader than just generalization. Fundamental shifts in perspective are beginning to appear among musicians and music theorists as a result of this influence. A comparison of the standard literature of music theory such as Piston [1978] (or the seminal works such as Fux [1965] and Rameau [1965]) with the formulations discussed in the survey section of this article show that radical changes have occurred. Classical music theory has been predominantly analytical. Its main use has been to explain practices among composers of a past era. In contrast, the practical need for formal descriptive systems demanded by computer music applications has stimulated the development of a body of *generative* music theory, which is much more tightly coupled with composing and realizing music.[1]

Historically, algorithmic procedures have always been fundamental to the creation of music. The rigorous application of algorithms based on an a priori theory of composition, however, seems to have first occurred in Western music in the twelve-tone serial compositional techniques of Arnold Schoenberg, Anton von Webern, and others in the first decades of the twentieth century [Schoenberg 1950]. In the 1930s and 1940s,

[1] We are using the term *generative* to describe a music theory which is sufficiently formal that it would be capable of actually generating compositions from the rules of the theory. We are not using the term with the strict meaning it has acquired in the school of generative-transformational linguistics [Chomsky 1965].

the theoretical writings of Joseph Schillinger outlined a mathematically rigorous music theory [Schillinger 1941]. Schillinger, who predicted the use of computers in musical composition, tried to develop a theory of music that would be of practical use to composers in their craft. In the late 1940s and 1950s, composers Karlheinz Stockhausen, Pierre Boulez, and others greatly extended the range of musical "parameters" to which the serial practices of the twelve-tone school were applied.

The first composer to use computers for compositional purposes was Lejaren Hiller, who in the late 1950s used a computer to generate the composition *Illiac Suite for String Quartet* using stochastic procedures derived from his work in chemistry, which were in turn derived from the field of mathematical linguistics [Hiller and Isaacson 1959].

General theories of music have recently been proposed that are developed from insights derived from other fields, such as cognitive psychology [Tenney and Polansky 1980]. The influence of the generative-transformational grammars developed in linguistics can be seen in the music theory of Lerdahl and Jackendoff [1983]. Musicologists [Lincoln 1970], music typographers [Byrd 1984], and music instructors [Gross 1981] have found computable formal systems to be a naturally ally. Music theory has itself been studied for an understanding of the cognitive processes in music [Laske 1980; Minsky 1981].

These observations demonstrate that the theoretical and compositional aspects of musical thought have been influenced in this century by the growing importance, complexity, and expressiveness of formal systems and formalized ways of thinking. The ground has been prepared for the application of computational paradigms, and computers themselves, to the evolution of musical thought and behavior.

The increasing interest and importance of formal thinking, in music or in any other area, cannot meaningfully be separated from the development of formal *languages* through which that thinking is officially expressed, and unofficially explained. In music, the active, explicit creation of formal languages to express aspects of theory, performance, or composition, did not come until the computer made it possible to mechanize some aspects of the processing of formal languages. Thus work carried out in the early part of this century, which we now recognize as having a strong formal bias, was expressed in terms of informal languages, or as systems of rules constraining the creation or analysis of a musical text. This work generally emphasized the limitations of a purely formal approach, insisting in various ways that some irreducible residue remained in the domain of creative, artistic (and ipso facto informal) endeavor.

The above remarks serve to point out the breadth of interest in computable musical formalisms by musicians and others. Because of this breadth, we must carefully delineate the scope of this survey so as to keep it within reasonable bounds. The principal razor that we shall use in this delineation is the term *programming language*, the meaning of which we shall now discuss.

Many musical applications of computers cannot properly be said to involve programming languages in any direct way. For example, a musicological application in which common characteristics of a large number of Renaissance melodic fragments are studied by pattern-matching techniques does not present the musicologist with a programming language. Similarly, a program that generates compositions automatically using pseudorandom number generators to drive hard-wired stylistic rules cannot properly be said to present its user with a programming language. Rather, in each case the programmer has used a preexisting programming language to create an application in which inputs of a simple structure produce effects (such as outputs) desired by the user, such as analysis results or compositions.

As the input to a program—and its response to that input—become more complex, it becomes less clear whether it is a programming language or not. It is difficult to give a categorical definition of the term that would clarify this. It would be well beyond the scope of this survey to motivate and justify such a definition. Nevertheless, it is intuitively clear that a large number of possible programs are associated with a

programming language, that these programs can exhibit a considerable degree of what we might call structural complexity, and that the processor that implements the language responds to each input program in a way that is at once fairly definite, and is systematically related to its structure.

The development of programming languages specifically for musical applications seems to have been most concentrated in the areas of sound synthesis and musical performance. The only other musical area in which programming languages figure strongly enough to warrant the attention of a survey is composition. As we shall only partially cover computer music composition systems, we must further delineate this area.

Three strategies have been commonly employed in developing compositional uses of computers. A composer experienced at programming might experiment by successively modifying a composition program written in an existing programming language [Barlow 1980; Koenig 1970a, 1970b]. The languages chosen vary widely, and also include generative grammars [Green 1980; Holtzman 1981; Roads 1985a]. Another strategy is to develop libraries of utility subroutines that implement common operations on musical data structures. Then composition programs can be written in some standard programming language that makes calls on the library subroutines. If the language can be interpreted, a fairly responsive interactive compositional system can result [Nelson 1977].

While neither of these approaches can be said to involve the practitioner in the development of programming languages, these methods have been used quite frequently by composers to embody particular compositional systems [Hiller et al. 1966; Koenig 1970a, 1970b; Truax 1977; Xenakis 1971]. The aesthetic and technical underpinnings of this class of work are extremely diverse, and in most cases not directly related to programming language issues, and so will not be included here except incidentally.

A third approach is to write a programming language as the embodiment of a musical paradigm through which a wide range of compositional strategies can be realized. Programming languages for music typically go beyond standard general-purpose languages in implementing special musical data types and operations, methods for representing the flow of time, and methods to represent the forms of recursion and parallelism commonly found in music.

Within this third category, however, we must discriminate between two categories of approach. There are, on the one hand, *descriptive* languages—used for music data input—and on the other, *generative* languages—used to compute musical scores from a program input. Most actual languages in this category have some component of each; they usually can be differentiated, however, by the intended application. Examples of pure descriptive languages used to capture existing music in manuscript form for music printing or musicological analysis include DARMS [Erickson 1975] and MUSTRAN [Wenker 1972]. Some descriptive languages provide simple compositional processing facilities, such as SCORE [Smith 1976] and YAMIL [Fry 1980].

Generative composition languages usually also come with descriptive musical data structures, but emphasize compositional processing. The range of available models to represent music data in this class of language is quite broad, and includes mathematical models such as stochastic and combinatorial techniques, linguistic models such as grammars [Roads 1985a], algorithmic models such as those found in typical high-level structured programming languages, process models such as those found in object-oriented programming and parallel processing, and other models derived from artificial intelligence [Roads 1985b].

We concentrate on programming languages that have been developed for sound synthesis, musical performance, and selected programming languages for composition. The music programming languages surveyed all have close ties to a synthesis system, and can be described as having strong generative capabilities. The rest of the survey is as follows. Section 1 contains an overview of traditional musical notation and practice, how it categorizes music, and

the issues it addresses. Section 2 is a brief look at some representational issues for sound and music on computers, in relation to traditional notation. Section 3, the main body of the paper, is a survey of several of the reasonably well-known computer music languages, and closes with a look at work being done currently. Although our survey is not exhaustive, we believe that it does cover a substantial cross section of the influential work in the area of computer music language design.

1. REPRESENTATION OF TRADITIONAL MUSIC

To gain insight into the issues involved in the design of languages for computer music, it is first necessary to understand what process is being modeled. So, by way of introduction, we briefly discuss traditional Western music performance and notation with an eye to the parts that would be required for a computer model.

Probably the most important observation to be made from our point of view is that traditional music notation is designed to serve the needs and processing abilities of humans; it bears little resemblance to a formal language. The degree of abstraction in CPN is such that it telescopes many levels of description together, resulting in a system that is extremely concise and effective, if formally rather incoherent.

1.1 The Musical Note

Performed music is a continuous stream of sound. Notational systems are by nature discrete, so capturing music in a notation first requires that the elements to be rendered are parsed out of the sound stream. The most evident abstraction device of CPN is the quantization of the continuous musical stream into discrete *notes*. Notes are event specifications that code pitch, onset time, and duration. They are indicated as large dots (the *note head*), with vertical *stems* placed on a longitudinal grid of five lines, called a *staff*. The shape of the head and the presence or absence of *flags* on the stem, or *beams* connecting stems, determine duration. Multiple-note heads can be grouped on a stem, producing a *chord*. Notes are sometimes connected by curved lines called *slurs* to show their grouping into *phrases*. The flow of time is indicated on the staff by the accumulated durations of the notes. Height on the staff determines pitch. Notes of the same pitch that are connected by slurs are said to be *tied*, meaning their duration is the compound of slurred notes. An example of CPN is given in Figure 1.

Where a continuum is quantized, information is necessarily lost. The trick that CPN accomplishes is to allow the information not carried in the score to be recovered through a set of implicit rules. Thus a satisfactory realization of an encoded work can be reconstituted through the *interpretive practice* of trained performers. The knowledge that enables human performers to interpret music notation is extremely difficult to represent in a formal way, although humans can be trained to use it very well indeed (but see Sundberg et al. [1983]). The concept of the musical note allows what we call the *principal perceptual dimensions* of music to be abstracted into a concise symbolic representation. It embodies a set of coding rules for musical scores that reduces the information content in the score to humanly manageable proportions. A performer can use the rules of musical interpretation to reconstitute an acceptable facsimile of the musical idea during performance. Evidence for the reasonableness of this explanation is the fact that synthesizing music from only the information manifested in a traditional score, which omits the information supplied by the performer, results in a wooden rendition which one strains even to call "musical." This is the effect given by music boxes and the mechanical organs of the past millenium, as well as by some of the crude computer music systems of today.

1.2 Principal Perceptual Dimensions of Music

The principal perceptual dimensions in music are the complex psychoacoustical domains of *pitch*, *rhythm*, *loudness*, and *timbre*. (Timbre can be characterized simply as "tone color," for example, "sharp,"

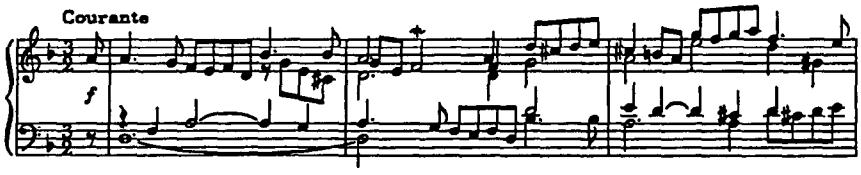

Figure 1. This example of common practice notation (CPN), which shows the opening measures of a harpsichord work by J. S. Bach [Bach 1742], was generated by Leland Smith's music manuscripting program, MS, developed at Stanford's Center for Computer Research in Music and Acoustics. The two rows of five horizontal lines are *staves*, which demark a grid with relative pitch on the y axis, and relative elapsed time on the x axis. The lower of the two staves is for low pitches; the upper is for high. *Notes* are event specifications that code pitch, onset time, and duration. They are indicated as large dots (the *note head*), with vertical *stems* placed at the intersection of the appropriate x and y locations. The shape of the note determines its duration, for example, if the note head is hollow or filled, if it has a stem, and if it has (one or more) *flags* on the stem, or is *beamed* together with other notes.

As CPN is basically an event-oriented notation, additional signs exist to indicate where the note events are to be *slurred* together into a continuous line. Slurs are also used to join notes of equal pitch into compound durations called *ties*. Only ties are shown in this example.

The word *Courante* indicates a particular dance rhythm and attendant feeling that the work is supposed to evoke. As described in the text, such terms are *saturated symbols*, which connote many performance indications simultaneously, and at many levels of meaning to the musician.

"dull," "brilliant." It is also used to describe the range of sounds obtainable with the different families of orchestral instruments.) In CPN, the first two dimensions are notated systematically, and loudness less so. Timbre also is not notated systematically in CPN, but is a by-product of controlling which instruments play, as well as what, how, and when they play. There are numerous other dimensions that are relevant to a musical work, such as the number of instruments of various kinds, and where they are physically located, but these are auxiliary in most music.

Translating even the apparently straightforward dimensions into precise physical parameters of frequency and sound pressure level is more difficult than it at first appears. In recent years, much psychoacoustical research has been addressed to this problem. Perceived pitch is, of course, strongly correlated with frequency of a periodic, or nearly periodic, signal, but it can also depend on amplitude and timbre. Timbre itself is a multidimensional quality [Grey 1975]. Regularizing its description has been the subject of much discussion [Balzano 1985; Shepard 1964; Wessel 1979].

1.3 Performance Interpretation

As we have stated, the information contained in a traditional score is interpreted in performance according to a set of rules that is formally incoherent, known as *performance interpretation*. Examples of these rules include *articulation*, the practice of joining or separating successive notes, *dynamics*, the pattern of perceived amplitude change through time, usually correlated with some measure of "forcefulness" appropriate to a given musical instrument, and many others.

1.4 Structural Organization

Traditional music admits various organizing principles, both hierarchical and nonhierarchical. An example of hierarchy is the notion of *phrase*—a sequence of notes forming a larger musical unit that usually forms a complete semantic object, although not necessarily a complete idea or statement. The musical phrase is related to the linguistic phrase, in that both can be traced to physiological roots (speakers and singers need to breathe). Other hierarchical groupings include motives (figures used repetitively, with or without transformations),

sections (groups of phrases forming a complete thought), movements (complete and independent divisions of a large work), and multimovement works such as suites and symphonies.

Another place where hierarchy normally occurs is in the organization of pitch scales used in CPN. There is typically a *pitch class* (i.e., one of the 12 pitches of the chromatic scale), which is taken as a *tonic*, and forms an axis for all other pitches of the scale surrounding it. Often, the shape of a work can be traced by the way the composer shifts the focus of which pitch is emphasized, or *tonicized*, as the work progresses according to strict hierarchical ordering procedures. In much music in the twentieth century this hierarchical organization of pitch has been thrown over in favor of other ordering principles, such as that in the *twelve-tone* music of Schoenberg [1950], or other pitch systems, such as the microtonal music of Harry Partch [1949].

1.5 Performance Directives

Notational conventions have evolved for instructing the players of particular instruments or families of instruments. For example, CPN includes marks indicating bow direction (for stringed instruments such as the violin), pedaling (for keyboard instruments such as the piano), fingering specifications, breath marks (for woodwinds and brass), use of mutes (for brass and stringed instruments), and more. This *operational* component of CPN represents a compromise between the desire to have a pure notation that specifies just the notes and one that makes room for nuances peculiar to certain instruments. These nuances are often very significant, helping in various ways to reveal more of the structure of a work, or to place it in a particular stylistic context, or to modify or reinforce a particular emotional tone, or to simply aid the performer in mastering the job of playing the piece. Neither traditional music theory nor computer-based music synthesis systems have succeeded thus far in modeling this aspect of CPN in any systematic way.

Abstract aspects of music, such as elements of muscial *style*, are not explicitly represented in CPN, although they can be easily deduced by trained musicians. This is clearest in the case of Baroque music (European music of the period 1600–1750), in which the operational component in CPN is very small; yet this music does not lack stylistic identity. On the other hand, modern styles of music often have such a burden of operational components that they require major extensions to CPN, or even its replacement by other notational conventions in order to express the intentions of the composer and/or the actions expected of the performers [Cage 1969].

1.6 Flow of Musical Information

In this section we discuss the process of communication among performers and audiences. Communication is involved even when a single player is practicing alone, for there is a part that plays and a part that listens and (we hope) criticizes. The following are some examples of the modes in which that communication is managed in traditional ensembles of musicians.

Symphony orchestras are organized in various overlapping ways. The canonical order is a tree-structured chain of command from conductor to leader of each section (or instrumental class: woodwinds, strings, brass, percussion), to individual players. In reality, there is considerable horizontal flow of information among sections and among individuals, guided as much by the inner organization of the music as by the conductor. The canonical order is further perturbed when a soloist is added, for example, in playing a concerto. On the other hand, the organization of a string quartet in performance is more fluid and less hierarchical, even varying with the style of music being played. For example, in a classical string quartet by Joseph Haydn, the first violinist usually plays a dominant role, whereas in a more recent quartet by Bela Bartók, leadership is passed among the players according to the musical context.

Examples of other kinds of interaction abound. Jazz, for example, often alternates between structured sections, where all the players follow a score, and solos, where one player improvises while the others accom-

pany. *Free jazz* involves the performers in direct, impromptu extemporizing on their accumulated musical knowledge. Composers and performers of the twentieth century have surpassed all previous generations in finding and developing alternate orderings of all kinds.

The performance of a musical work reconciles all these interaction strategies in real time. This is difficult, even for highly trained musicians, and it is common for professional musicians to spend weeks rehearsing a new work. This applies to jazz as much as to classical music.

1.7 Music as a Knowledge Representation Problem

The deep cultural heritage embedded in traditional music could yield some insights about the nature of our own cognition. For instance, information derived from the structure of a generative theory of music could be correlated with a theory for language to gain a broader perspective on the fundamental mechanisms of human communication and understanding. This has been taken by some computer scientists as an exciting opportunity to explore music as a knowledge representation problem. Practical tools to study the problem must be available, however, before any comprehensive representation of music (as opposed to just the objective dimensions of it) can be made. This focus on practicality also has the advantage of addressing the needs of musicians and composers in their attempts to realize the promise of computers for music.

1.8 Summary

Common practice notation is surprisingly compact in view of the objective complexity of what it is modeling. There are two reasons for this. First, CPN reduces the principal dimensions of music to pitch and time. Second, the remaining dimensions (dynamics, timbre, and articulation) are rendered by symbols and icons that have evolved over hundreds of years of performance practice. When these fail, CPN resorts to exhortations in one of several natural languages (usually Italian). Thus CPN presents a musical artifact in a form that mixes objective dimensions such as pitch, time, and dynamics, with deeply embedded, and often unarticulated, knowledge of musical style and history.

The rest of this article deals with formal, operational notations designed to solve musical problems with computers, and their implementation. The perspective that we obtain by looking at CPN, and the general issues of musical performance, allow us to see the larger context in which these systems operate.

2. DIGITAL REPRESENTATION OF SOUND

The theory of digital sound representation is based on the theory of sampled data functions, which was originally worked out by Nyquist and others in the 1930s and 1940s [Bennett 1948; Ragazzini and Franklin 1958] and first applied to the digital transmission of speech in the 1950s [David et al. 1958]. The musical application of this technology is described by Mathews et al. [1969].

For the purposes of this survey, suffice it to say that sound can be represented in a computer memory as a stream of air pressure wave quanta, called samples. The samples constitute a periodic function of time, describing the fluctuations of the air pressure wave that correspond to the sound. A sound can be recorded by digitizing an electrical signal derived from a microphone through an analog-to-digital converter (ADC). Correspondingly, a digitized recording can be played back through a digital-to-analog converter (DAC), which produces an analog voltage suitable for audio reproduction equipment. In digitized form, recorded sounds can be analyzed and processed, or arbitrary pressure functions can be synthesized.

Whereas CPN is largely a symbolic representation for the resulting sound, the digital encoding of sound as pressure functions is iconic in the sense that the function resembles the sound being represented (the closer the resemblance, the greater the fidelity). Most music synthesis languages provide both functional and symbolic no-

tations, since the resulting musical meaning often seems better expressed. CPN carries an iconic operational component for the same reason. Sometimes, a variety of symbolic systems will be required to represent the multifaceted meaning of a composition in a music programming language. We shall see examples of this problem in our discussion of the language Pla. Synthesis languages deal with the translation of symbolic and/or iconic representations of musical sound into a pressure function, or into a stream of machine instructions that, when processed, yield such a function.

After the technological issues for representing speech were adequately addressed, the same techniques were applied to the production of music. Researchers at AT&T Bell Laboratories at Murray Hill, New Jersey, led by Max V. Mathews, developed what they called an acoustic compiler in the late 1950s and early 1960s [Mathews 1961]. This research led to a suite of computer music languages, discussed more fully in the next section. These languages were outgrowths of programs developed for analog circuit modeling by digital computer [Mathews 1963]. Just as the first uses of digital computers in signal processing were to model analog circuitry [Rabiner and Schaffer 1978], the first computer music systems modeled the analog electronic music systems being used at that time [Roads 1980b].

The basic ordering principles of digital music synthesis and processing can be seen as extensions of those principles used in analog electronic music. When first developed in the late 1940s, the most salient feature of electronic musical instruments was felt to be their ability to allow musicians to dig into the acoustical microstructure of sound. This allowed the musical dimension of timbre, which was previously constrained by the sonic limitations of traditional musical instruments, to grow into a major dimension of musical interest and concern. This corresponded to the wishes expressed by many composers to expand the importance of this dimension in their music [Bussoni 1962; Cage 1961; Risset 1969; Varèse 1971]. In some cases composers treated all sound as potentially musical. An example of this is *musique concrète* [Schaeffer 1952].

The interest in control of timbre was a primary impetus in the growth of electronic music. The severe liabilities of analog technology limited its actual usefulness to music, however. The use of digital computers represented a clear gain over analog techniques in terms of precision, repeatability, and micro- and macrolevel control. Nonetheless, computer music systems inherited many of the notions of analog music systems, such as oscillators, envelope controllers, filters, amplifiers and recorders, many of which, in turn, had been originally borrowed from the electronics shops of early radio stations.

In keeping with the rise in importance of timbre among composers of this century, most computer music notations define a musical note as the *specification of an acoustic event.* That is, a note consists of a collection of acoustic parameters that are applied to a synthesis procedure of some sort that generates an acoustical waveform matching the specification. Clearly, this is quite different from CPN, where a note specifies a human gesture toward an instrument, the secondary result of which is the excitation of an acoustical waveform determined by the combination of gesture and instrument.

By going directly from the note specification to the acoustical waveform, digital music synthesis provides a very precise means of stipulating the kind of signal desired. This is both an advantage and a liability. It is an advantage insofar as it allows composers freedom of expression far beyond that available from traditional musical instruments. With digital music synthesis it is possible to create sounds that no physical body can produce (other than a loudspeaker, of course), or transcend the capabilities of human performers. It is a limitation insofar as the composer must now fill in performance detail in musically interesting ways. In traditional music, the instrument and the performer take care of sounding musical; the composer can rely on it. In computer music, the composer must be, as it were, an "instrument designer" and "performer" as well.

3. SURVEY OF LANGUAGES FOR COMPUTER MUSIC SYNTHESIS

We now turn to a survey of programming languages developed for particular applications in computer music. This survey covers software sound synthesis, languages for hardware synthesis control, real-time control of hardware synthesis, and high-level programming languages for musical composition. This survey is far from exhaustive; we have presented only a few of the languages that address a particular application area to conserve space.

3.1 Software Sound Synthesis Systems

Language design for digital music synthesis has not developed around a single theme, but it is useful to see the single model of *note as acoustic specification* as a starting point, with higher level representations gradually added. As we show in the following survey, early languages focused on flexible and efficient methods of signal processing for the *note as acoustic specification* paradigm, and provided only rudimentary tools for composers to specify scores. Composers were required to work outside the domain of these languages, at a higher level to compose musical notes or a lower level to directly manipulate the acoustic waveform. Some research was directed toward higher level compositional expression, such as developing synthesis language preprocessors [Smith 1976]. Others sought to work directly with the microstructure of the sonic waveform by applying compositional rules—as opposed to rules derived from acoustics—to the generation of waveforms [Blum 1979; Xenakis 1971]. As the ideas of *note* and *acoustic specification* are largely irrelevant here, this work is an example of an alternative approach to the one we discuss.

As things have progressed, synthesis language developers have focused increasingly on representing higher level musical structure in terms of the original *note as acoustic specification* model. This has brought us full circle to a resurgence of interest in music theory, but from a perspective resulting from two major advances in our understanding. First, we are more knowledgeable about the acoustical behavior of musical signals, and second, we are in a better position to reexamine the formal boundaries and map the uncharted areas of music.

3.1.1 Music N

As mentioned in Section 2, the first programming languages for musical sound synthesis were a suite of programs, named Music I through Music V, developed at AT&T Bell Laboratories, which we refer to collectively as Music N [Mathews 1961, 1963; Mathews et al. 1969]. This early work spawned numerous lexical descendants, and recent designs still emulate important features of these languages. We focus our attention on the last of the suite, Music V [Mathews et al. 1969].

Music V was designed to run in an operating environment that includes a high-speed general-purpose computer, a large mass-storage device to hold synthesized acoustic waveforms, and high-quality digital-to-analog converters (DAC). Music V operates as follows: It accepts input that contains instructions about synthesis algorithms and a particular musical score, generates the samples for the entire score, and stores them on disk or tape. Then, an auxiliary program is used to read the samples off the disk (usually in real time) and send them to one or more DACs. (Extensions have been made at some sites to allow processing of sound recorded via ADCs as well.) We call this *off-line* synthesis. In such a method, no real-time interaction takes place between a composer or performer and the synthesis program. Interaction is limited to stating and stopping data conversion.

In developing the language, the major problems have been to find a way of dealing efficiently with the computational requirements of the music synthesis problem, and to find a design whose expressive capabilities could combine simplicity, uniformity, and power. The solution for these problems as developed by Mathews consisted of a threefold approach:

(1) Synthesis algorithms are specified as combinations of building blocks called *unit generators*. A unit generator is a predefined

synthesis module that performs some simple signal-processing function, such as oscillation, gain scaling, or filtering. An *instrument* is a named collection of unit generators, configured into a data-flow subprogram.

(2) The activation of instruments is controlled by *note* statements. A *score* is a collection of note statements and instrument definitions.

(3) Efficiency needs are addressed by localizing the computational load into unit generators, and by emphasizing the use of lookup tables and simple transforms, especially for such things as generating oscillations and applying amplitude or frequency envelopes to them.

3.1.1.1 Efficiency Concerns. In order to assess the strengths and weaknesses of this design, we need to look more carefully at the forces that created it. Probably the most important of these is the problem of computational efficiency. The Nyquist sampling theorem states that the sampling rate must be (at least) twice the highest frequency desired. As the ear is capable of hearing up to 20 kilohertz (kHz), this implies a sampling rate in excess of 40 kHz, that is, 40,000 discrete samples per second per channel. (Currently favored standard sampling rates are 48 kHz and 44.1 kHz.) Each sample of a complex synthetic waveform can be the result of thousands of machine instructions, including a large percentage of multiplication (which is one of the most time-consuming instructions on most general-purpose computers.) In the early days of computer music, a sampling rate of 20 kHz was typically used because computer systems simply did not have the resources to operate at more adequate rates.

Indeed, computational efficiency drove many aspects of earlier computer music research. One of the major activities in computer music research has been the identification of algorithms that allow the synthesis of a wide variety of interesting sounds in a computationally efficient way.

In software sound synthesis, a useful measure of the efficiency of a program on a given input is its *compute ratio*, that is, the ratio of the elapsed time required to compute 1 second of synthesized sound. A ratio of 1 or less indicates that the sound is computed in real time or faster. Such a compute ratio precludes the need for off-line synthesis, and also foregoes the need for large mass-storage systems, since waveforms can be generated "on the fly." Composers using programs such as Music V to create even moderately sophisticated music, however, continually demonstrate that even the fastest general-purpose computer systems cannot achieve a compute ratio of 1 or less. In fact, even with considerable optimization, compute ratios higher than 200 to 1 are common. At the level of 60 to 1, an hour of compute time is required to compute each minute of sound. Obviously, the computational requirements (not to mention the storage requirements) of digital audio can exceed the performance of the most robust general-purpose computer systems available. Although there are some clever algorithms for efficient computation of interesting sounds, general-purpose computers are not an appropriate vehicle for real-time synthesis. The price of generality is processing speed degradation. For many signal-processing tasks, direct hardware implementation of the processing performed by unit generators is a more sensible approach.

3.1.1.2 Unit Generators. A unit generator is a predefined synthesis module that performs a simple signal processing function such as oscillation, gain scaling, or filtering. In this respect, it resembles the analog electronic synthesizer module, which, in turn, is an abstraction from modular electronic test apparatus familiar to electronics engineers. Examples of unit generators include oscillators, filters, function generators, and mixers as well as simple unit generators to do arithmetic and conditional testing of signals.

Complex synthesis algorithms are built up by combining unit generators. The output of a unit generator is stored in a labeled scratch-pad memory, which is then available as the input to any other unit generator for further processing. The resulting composite synthesis algorithm is a data flow subprogram called an *instrument*, which is executed to produce the final

waveform. Not surprisingly, a collection of these instruments is called an *orchestra.*

The design of Music V addressed efficiency issues by separating synthesis into basic operations with flexible means of interconnection. Oscillators use memory-resident lookup tables to generate any periodic waveform cheaply. Simple combinations of linear ramping functions are pieced together into efficiently computable envelopes and control functions. Music V has relatively few unit generators, so that in most implementations, the code for the unit generators can be assembly coded easily for maximum efficiency.

The genius in the design of Music V is that it takes a middle ground between a high-level "hard-wired" approach and a low-level "roll-your-own" approach to the integration of its various components. The former approach might simplify programming, but it would be at the expense of the language's general expressive power. The latter approach would overwhelm the user with meaningless possibilities. The important place that Music V has held over the course of time results in no small part from its adherence to this important middle level of integration.

3.1.1.3 The Note Statement. In Music V, an instrument is invoked when a note statement is encountered. A note statement consists of the keyword "note," followed by a list of expressions, which are parsed by the Music V interpreter. The first three of these expressions have a fixed interpretation; they designate the instrument begin time, name, and duration. Each note statement causes an instance of the named instrument to become active at the specified begin time, and to remain active during the specified duration. Subsequent expressions in the note statement are used to pass parameters to the instrument to control its synthesis algorithm.

Music V has a very limited ability to interpret arithmetic expressions in note statements. Even the method for converting frequency in hertz to the value used by Music V oscillators to compute that frequency must be supplied by linking subroutine modules into the Music V executable image. On the other hand, the mechanism of linking special routines into the binary image is quite powerful (if rather low level) and is used to link simple composing subroutines that create or modify certain parameters of scores.

3.1.2 Music IV and Music V

Although we have focused on Music V, certain of the important variants described in the next section were derived from its predecesor, Music IV. Both programs were implemented as three-pass systems. Pass 1 performs parameter conversion, Pass 2 sorts scores according to their begin times, and Pass 3 does the actual synthesis.

Music IV lacked reentrant instruments, a feature that was added in Music V. A code module is said to be *reentrant* if multiple instances of it can be active at the same time. This can be achieved if the module is a *pure procedure,* that is, if the module consists only of program instructions and constants, using separate areas of memory for storage of (possibly multiple copies of) variable data that is to be unique to each instance of the module. The chief benefit of reentrancy comes when generating sounds to be heard simultaneously, such as chords. In Music IV, one would need as many separately defined instruments as there were notes in the densest chord in a piece, even if they all were to produce the same sound. Music V allowed a score to simply refer to an instrument as a template, which was then instantiated as many times as necessary to produce the chord.

Music IV computed samples by parsing the instruments into an execution tree, which was traversed once for each sample. Music V instruments were parsed into a block-at-a-time execution tree. Music IV's method was more flexible in terms of program flow, in that the execution tree could be modified for each pass through the tree, allowing the synthesis operation to change for each sample, if necessary. It was less efficient, however, since operands for unit generators were usually used only once per pass, resulting in much wasted access to main memory to fetch and store them. Music V corrected this inefficiency by computing a block of samples for each unit generator on each pass. This allowed the

computer to use the operands loaded into high-speed accumulators repeatedly before having to fetch the operands for the next unit generator.

Music V has been a remarkably successful language, both in number of installations and influence on successors. Certainly, one of the reasons for its widespread use is portability. Music V took advantage of what was at the time the recent advent of machine independent languages: it was written mostly in FORTRAN IV, and ran originally on a General Electric 635 computer. The unit generators were written in assembly language, but even so, the program was relatively easy to transport. A number of variants sprang up at different computer centers where the appropriate technology was to be found.

3.1.3 Variants of Music N

Among the Music IV variants developed in the late 1960s and early 1970s were Music IVB and Music IVBF, written by Hubert Howe and Godfrey Winham [Howe 1975]. Music V variants included Music 360 and Music 11 for the IBM 360 and the PDP 11 computers, respectively, written by Barry Vercoe, Roger Hale, and Carl Howe of the Electronic Music Studio at Massachusetts Institute of Technology [Vercoe 1979]. The most recent implementation of the Music V model is Cmusic, written in the C programming language [Moore 1982]. Although each of these implementations added numerous useful features, they differ little in substance from their model. Music 11, for instance, introduced control flow statements in instruments similar to FORTRAN **if** and **goto** statements. Cmusic is perhaps the purest implementation of its model in terms of fidelity to Music V program syntax, but its operation has also developed beyond its Music V model to make its use more flexible. Figures 2, 3, and 4 are a complete Cmusic score, which illustrates the declaration of instruments, time domain functions, and a simple score consisting of one note.

The most radical version of Music IV is MUS10, developed originally at Stanford by David Poole, to run on a DEC PDP-10 computer [McNabb 1981; Tovar and Smith

```
ins 0 FM;
osc b1 p9 p10 f2 d;
adn b1 b1 p8;
osc b1 b1 p7 f1 d;
adn b1 b1 p6;
osc b2 p5 p10 f3 d;
osc b1 b2 b1 f1 d;
out b1;
end;
```

Figure 2. Example of a Cmusic instrument definition that defines a data-flow program for frequency modulation (FM) synthesis. A block diagram of this instrument is given in Figure 3. See Chowning [1973] for a description of the use of FM in sound synthesis. Each line denotes a single unit generator (UG), named by its first field, as follows: **osc** is an oscillator, **adn** is a multiinput adder, and **out** is a data sink that writes sample streams to a disk file. The name of the instrument is **FM**, and the definition is terminated by the **end** statement. Fields on each line following a UG name are arguments that determine the source and destination of signal streams and controls for each UG. The notation **b1**, **b2**, and so on, denotes numbered data blocks, which are used to communicate data between UGs in a manner reminiscent of "patch cords." The notation **p5**, **p6**, etc., denotes parameters derived from **note** statements. The **note** statements can be thought of as "subroutine calls" to instruments, where the arguments to the **note** statement are stored in the sequentially numbered **p** fields. Collections of **note** statements are called **scores.**

1977]. MUS10 implements a substantial ALGOL-style programming language as well as providing unit generators. Instruments could be written from a combination of high-level programming language statements as well as unit generator calls. This facility allows composers to create their own unit generators in a high-level language, and also increases the intelligence an instrument can display toward information that it receives from note statements and from its environment, including other executing instruments.

MUS10 was implemented with the same concern for efficiency: It compiles an instrument definition to PDP-10 machine code, which is then run to generate the samples. The most compute-bound algorithms were put into the unit generators in hand-coded assembly language. Another extension of MUS10 is its introduction of **I_only** code, which is part of an instrument definition, but is run only before the first sample of each note. **I_only** code can perform functions such as those in Pass 1 of

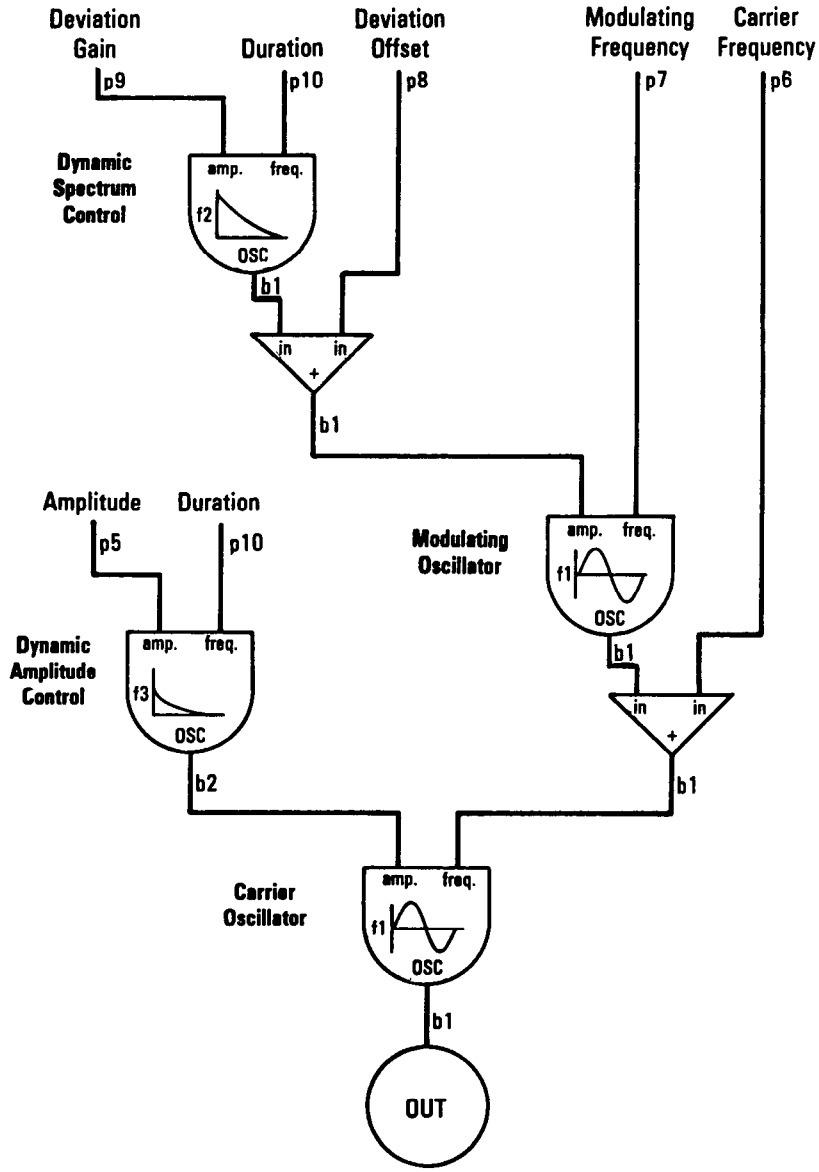

Figure 3. Block diagram of the Cmusic instrument definition given in Figure 2.

Music V, or more interestingly, it can be used to "interpret" its note statement to allow for slurs and glissandi across note statements, or coupling broader structural elements in the score.

It is worth pointing out that the starting point of MUS10 was an existing ALGOL parser, modified for music synthesis. We shall see several examples of this later in which the language designer simply took an existing language compiler and modified it to suit musical requirements. This is a very simple but effective way to start a language design.

```
gen 0 gen2 f1 1 1;
gen 0 gen4 f2 0 1 -1 1 0;
gen 0 gen4 f3 0 1 -1 1 0;
note 0 FM 15 1 200 Hz 280 Hz .01*p7
  10*p7 p4 sec 0;
ter;
```

Figure 4. This fragment of a Cmusic score shows **gen** statements used to create time-domain functions, and also shows the invocation of one instance of the instrument via a **note** statement. The **gen** statements declare and initialize time functions with names such as **f1**, **f2**, etc. The parameters specified in the **note** statement are applied to an instance of the named instrument. The first field, containing the word "note," is called **p1**. The second field, **p2**, holds the starting time in seconds of this instrument invocation; **p3** holds the name of the instrument to be invoked; **p4** holds the duration of this run of the instrument in seconds. Parameters **p5** through the end are handed to the instrument as implied arguments, and are used to pass parameters that control the data-flow program that the instrument implements.

In this case, the combination of the percussive shapes of the time functions created with the **gen** statements and the particular values passed in the **note** statement result in a bell-like tone that lasts 15 seconds.

3.1.4 Critique

The principal advantage of the Music N paradigm is the flexible and yet efficient method of synthesis specification via stored functions and unit generators. Another advantage is the homogeneous, nonhierarchical event specification via note statements. Where no hierarchy superimposes itself, the musician is free to superimpose structure on the stream of sound as desired, within the constraints of the note concept itself. This lack of structure is very attractive to composers attempting to forge new approaches to composition. It can also be seen as a liability, as we discuss later in this section.

Music N synthesis languages still impose the constraint of the "note as acoustical specification" model, even though event specifications are unconstrained. As we mentioned at the start of this section, there are those who would like to circumvent acoustical modeling, and directly control the acoustic waveform according to strictly compositional notions. PILE is an example of a music programming language that addresses these concerns [Berg 1979]. In PILE, no acoustical model is supposed. The usual parameters of synthesis are not implicit, but are constructed by writing programs in PILE. In a sense, PILE is equivalent to a programming language for implementing unit generators, although it is generally not used for this. PILE is usually used for direct application of algorithmic operations on the waveform, not based on any acoustical model.

The most significant limitation of Music N languages is that they do not run in real time. It is stifling to composers to be unable to hear near-term results. Some counter this complaint by arguing that composers of symphonies, for example, are often forced to wait weeks or months to hear their works. Although this is a complex topic, it is not difficult to guess what choice most composers would make between an off-line synthesis program and a real-time one of equal power. In addition, computer music has fewer boundaries on what sounds can be constructed, which makes tightly coupled interactive feedback more important. Even composers of traditional music could, and probably will, benefit from such a capability.

One aspect of music that is left out of all non-real-time synthesis languages is performance interpretation. There is very little facility for composers to shape phrases in musically characteristic ways on computers that do not allow the capture and study of actual musical gestures, which requires real-time data acquisition and control systems.

Another liability, discussed previously as an advantage, is the homogeneous, nonhierarchical specification of note statements. Two implicit assumptions of the Music V model of notes are as follows:

- All the relevant control information about a note is available when it is started (since parameters are passed only when the note is instantiated).
- Distinct notes can be treated as independent entities.

In truth, musical notes tend to have a life of their own, especially with instruments like the violin, where bowing and vibrato may change subtly throughout the note.

Also, the relationships between notes are many, and are of great musical significance. It is possible to model these relationships by adding a large quantity of global knowledge to the environment to be consulted by synthesis instruments, but this is not always a very convenient or natural way to do it.

Most of the languages described above are still in use, although many of them are reaching the end of their life cycle as the systems that supported them become obsolete. We must be careful, however, to make the distinction between the obsolescence of particular implementations of the Music N paradigm and the paradigm itself. At this point there are no challengers in the arena of software sound synthesis to the dominance of the Music N approach, only variations on its theme. Instead, as we see next, most development has come in the realm of hardware sound synthesis systems, and the development of higher level representations of music.

3.2 The Transition to Real-Time Synthesis

The liabilities of software sound synthesis mentioned above did not go unnoticed. The processing requirements of purely digital synthesis in real time made it economically infeasible until recently, however. One of the intermediate steps taken was called *hybrid synthesis*, which means synthesis using a combination of analog and digital techniques. Most commonly, this means analog sound synthesis with digital control, but sometimes the sound synthesis is partially digital.

3.2.1 GROOVE

One of the best-known and successful hybrid synthesis systems was developed at AT&T Bell Laboratories by Max Mathews and F. Richard Moore. Their system, called GROOVE, was based on digital control of analog synthesizers [Mathews and Moore 1970]. The first version of GROOVE ran in 1969, and it continued to evolve throughout the early 1970s. GROOVE consisted of a Honeywell DDP-224 computer with a bank of twelve 8-bit and two 12-bit DACs controlling a set of standard commercial synthesizer modules. In addition, it contained a variety of graphical and tactile control systems.

As with Music N, GROOVE skirted the problems of music representation by viewing music from the perspective of the system's implementation. Music was described as a set of time-ordered control functions that represented the performer's gestures. These functions (or others derived from them) were used to drive the converters, and thus control the synthesizer modules. There was no attempt to abstract the functions into higher musical terms such as notes. This approach provided the advantage of an unflavored environment in which to experiment, which was surely its primary goal. GROOVE could be used in real-time contexts, thereby allowing inquiry into performance practice. Its main limitation was the comparatively poor sound quality from the analog synthesizer and the limited control over it.

Various commercial systems have continued to extend the hybrid synthesis approach, typically using analog techniques wherever digital multiplication would have been called for (such as gain scaling and filtering) and using digital techniques everywhere else. Useful and cost-effective as such systems are, it is generally agreed that entirely digital systems are to be preferred because they are more stable, precise, and controlable.

3.2.2 MUSBOX

Specialized processors for digital sound synthesis began to appear in the mid-1970s. One of these early machines was the Systems Concepts Digital Synthesizer built by Peter Samson [Moorer 1981; Samson 1978]. The Center for Computer Research in Music and Acoustics at Stanford University commissioned the prototype and integrated it into a large time-sharing system (TOPS-10 on a PDP-10). The synthesizer has an instruction set that directly implements digital oscillators and other relevant operations such as filtering, signal testing, delay, and reverberation. Architecturally, the machine is highly pipelined, and uses time-division multiplexing through a sum-

ming memory to present the user with up to 256 digital oscillators and 128 signal modifiers, depending on the sampling rate. It is interfaced to the controlling computer via three direct memory access (DMA) channels, one for instructions and one each for input and output of digital sound.

Even though this synthesizer provides sufficient quantities of computation in real time to be able to realize sophisticated musical compositions, there are several reasons why no serious real-time control system has evolved for it. For one thing, it runs under a time-sharing operating system with limited real-time response. Subtle issues in the machine's architecture also made impromptu control in real time impossible for all but the simplest tasks. It was also considered important that the synthesizer provide backward compatibility with existing software (MUS10). Thus a compiler was written for it that simulated MUS10, but generated synthesizer instructions instead of directly producing a pressure function. The stream of commands was stored on disk and later fed to the synthesizer in real time.

The compiler was called MUSBOX, and it was written by one of the authors [Loy 1981] in the SAIL programming language [Reiser 1976]. MUSBOX utilized the parallel-processing simulation facilities in the SAIL language to model the parallelism in the synthesizer. In this programming paradigm, concurrently executing instances of subroutines were called *processes*, and each process resembled an autonomous program running "simultaneously" with other processes, and communicating via arguments and a message-passing discipline.

The main MUSBOX process contained an interpreter for Music N style scores. The "instruments" were modeled as processes that could be multiply instantiated. A time-ordered queue of runnable processes was maintained by a monitor process, and used by all processes to control the execution sequence. The interpreter process would read note statements and instantiate the correspondingly named instrument processes. These processes then would queue themselves to wait until the action time stated in the score. When awakened, an instrument process would analyze its note parameters and call low-level routines to compile synthesizer commands. It could also dynamically get information from other hierarchical levels and tune its performance to its environment.

This method proved very effective, in that it utilized a major high-level language, allowing considerable expressive power. The SAIL parallel process facilities proved to be useful for modeling not only the state of the synthesizer, but also the musical state. The notion of processes as autonomous entities, communicating and synchronizing with other entities toward a common goal, worked well as a model of the musical task. Behaviors could be encapsulated in the instruments in a way suggestive of musical knowledge, in the sense that an instrument could interpret its note statements in a characteristic way and could communicate dynamically with other instruments with a message-passing discipline.

3.3 Programming Languages for Real-time Hardware Synthesis Control

Before discussing individual solutions to the real-time synthesis control problem, we briefly survey some of the most important design considerations.

3.3.1 Dealing with I/O

The canonical use of a real-time musical sound synthesis system is to have one or more musicians controlling a digital musical instrument in performance, although many other modes are possible. Categories of control inputs include standard musical instruments to which sensors have been attached (such as pianos with switches on the keys), nonmusical input devices (such as knobs, slide potentiometers, switch matrices, and joysticks) and acoustic sources via microphones, and prestored sounds and scores.

The synthesis system is called on to interpret these inputs in many different ways to affect how the sound is synthesized. The strategy for dealing with I/O varies with

the nature of the data and bandwidth. The first consideration in such a design is to consider waveform sampling rate; currently, 44–48 kHz per channel is considered to be an adequate sampling rate for audio. This consideration affects the design of the synthesis hardware and limits the amount of computation available for each sample. Although this data rate is not hard to achieve with modern technology, the problem grows with the number of channels of synthesis output. A digital mixing system supporting as few as 8 channels (professional mixers often support as many as 24 or 36) will press the technological limitations of current data bus and disk technology.

The gestures of the controlling musician(s) must be received and analyzed. The rate at which these inputs are scanned must be fast enough to not miss events. A rate of 1000 Hz is adequate for most cases. Slower rates (down to 60 Hz) seem adequate for some purposes. This is the per-channel rate. Typically there is a larger number of control input channels than of sound output channels, even if not all of the input controls are in active use at any one time.

Another consideration is that some performance variables may change more or less continuously, whereas changes in others will be strongly correlated with note boundaries, which are irregularly spaced. This produces a very spiked information density curve. As a consequence, the bandwidth of information on the control channels is often quite irregular. Thus it seems that the bandwidth requirements for completely adequate gesture capture might come close to that required for audio data.

Designing a real-time system to deal with this information is difficult for two reasons. First, there is a lot of information. (There can be thousands of times the amount of interim information as there is final musical signal.) Second, as we have pointed out earlier, the information is often used in sophisticated ways. That is, it may require considerable processing to get from a control input, viewed as a function of time, to the related function that is of actual acoustical, or even musical, significance.

3.3.2 Response Time

The "bottom line" of real-time performance requirements is that delay in response to performance gestures should be some minimal, predictable function of time. That is, two criteria apply. The first is that the delay from initiation of a performance gesture to the resultant sound should be perceptually brief. For traditional instruments these are on the order of a few milliseconds for percussion instruments, 30–50 milliseconds for some strings and brass, and up to 1 second for pipe organs in large rooms.

Second, and more important, whatever delay exists must be predictable so that the performing musician is able to track the underlying rhythmic pulse of the music. In particular, the response time must not be a function of the density of events, since delays of this sort do not normally occur in orchestral instruments. For instance, it should not take perceptually longer for many notes to start up than for just a few, as this would adversely affect the performer's ability to synchronize a performance. Computers that are multiplexing tasks have a liability to this kind of degradation, and considerable care must be exercised to avoid it.

3.3.3 Periodic versus Event-Driven Modeling of Musical Processes

Another important system design consideration is the underlying model of computational, acoustical, and musical processes. Some kinds of musical processes are best monitored on a periodic basis, whereas others are best handled as discrete, asynchronous events. Periodic monitoring is usually implemented by having a processor poll the state of the various inputs in some order; discrete events are usually implemented by interrupting the processor when they occur.

There is a fundamental dichotomy between polled and event-driven systems. By their nature, polled systems are best for representing continuous functions of time, but event-driven systems provide a built-in criterion for parsing music into discrete quanta whereas polled systems do not. On the other hand, the time overhead associated with interrupt processing means that

the amount of available computer power to process interrupts is an inverse function of the number of interrupts. This can be dangerous in live performance situations if it means that there are times when the interrupt processor can be so swamped that it cannot run in real time.

It tends to be easier to follow program execution in polled systems. On the other hand, modeling sparse events, such as note attacks with time functions, to the resolution necessary for these events to appear continuous to the ear, can be wasteful. Event-driven systems only respond to significant events (by definition), and tend to produce more compact (and therefore easier to digest) representations of the performance inputs. Also, event-driven systems offer a simple relationship between a stimulus and a response; polled systems have no such implicit relationship.

Some systems are either purely polled [Mathews and Moore 1970], or event-driven [Abbott 1981], or they may be combined in some way [Dannenberg 1984]. When they are combined, the low-level, signal-oriented parts of a system are most often modeled as (sampled) functions of time, whereas higher level parts, like the score, or gestural controls, are event driven. To see why, consider that the advantage of an event-based system is generally that there can be a simple expression for the relationship between stimulus and response. An example would be the relationship between the pressing of a key and sounding of a note, which is a relatively high-level and sparse occurrence in a synthesis system. The sparseness of the events means that the uncertainty associated with real-time operation of interrupt-driven systems is not an issue. On the other hand, the signal-oriented parts of a system are usually occupied with driving DACs that require strict periodic service. This naturally suggests a sampled function representation of signal-level processes.

3.3.4 PLAY

One of the early real-time control languages is PLAY, developed in 1977 by Joel Chadabe and Roger Meyers at the electronic music studio at The State University of New York, Albany [Chadabe and Meyers 1978]. This language has existed in several versions, first for an analog synthesizer, like GROOVE, and later for a digital synthesizer built by New England Digital (NEDCO). This synthesizer implements a fixed set of oscillators, which can be interconnected in a small number of ways. Thus most of the programming of this synthesizer is done by changing parameter values of given hardware unit generators, rather than creating and programming new synthesis algorithms. The control model of PLAY is very close to that of an analog electronic synthesizer, and provides an interesting example of the result of extending the models of analog synthesis to the digital domain.

PLAY is formally viewed as an interpretively implemented data-flow language, with operators to read streams of user-defined values, generate random numbers, or read values from real-time control inputs. In addition, any given version of PLAY has a set of available sound synthesis modules that can be controlled using these operators, and each other. The boundary between the control language and the signal processing sublanguage is not distinct, however, because everything is based on the same data-flow metaphor. Naturally, any given implementation will have limitations on what can be plugged into what. This defines the boundary between control and signal-processing levels for that implementation.

The pure data-flow metaphor is not enough for a synthesizer control language; a means of controlling the relationship of actions to time is also required. In GROOVE, this relationship is constant, in the sense that all functions are viewed as uniformly sampled. PLAY generalizes this by allowing modules (i.e., data-flow operators) to be *clocked* from a system timer, or from some other module that may generate evenly or unevenly spaced events. Indeed, any sequence of values may be translated to the time domain, so that any rhythm may be easily generated.

The interpretive nature of PLAY makes it possible to change definitions of user-

defined lists of values, and module interconnections in real-time, during execution. This facilitates "tuning up" a "score," which is really a program in the PLAY language.

To date, implementations of PLAY have been oriented toward relatively small computers and synthesizers, which has kept the user interface for definition of programs simple, in order to leave memory for real-time functions and for the scores themselves. In particular, the ability to remember gestures made by a performing musician, and later to analyze and/or edit them, is very hard to fit into a small machine, and is missing from PLAY.

3.3.5 4CED

4CED is a small programming language developed by one of the authors to control an experimental synthesizer at the Institute de Recherche et de Coordination Acoustique/Musique (IRCAM), in Paris, France [Abbott 1981]. The 4C is a small synthesizer implementing a set of fixed unit generators—oscillators, multiply/adders and envelopes—which can be arbitrarily interconnected via a set of scratch-pad registers [Moorer et al. 1979]. 4CED operates as a coalition of three loosely integrated sublanguages: an instrument definition language based on a set of unit generators, an integrated sublanguage for scores, and an interactive command language.

4CED's synthesis sublanguage is extensible in the sense that one can define "high-level unit generators" in terms of the basic ones (and other previously defined ones). These are implemented on the 4C by a macro expansion technique that handles allocation of computing resources and scratch-pad registers, and "dynamic binding" by interconnecting unit generators through scratch-pad registers. Since the synthesizer provides some hardware concurrency, the synthesis sublanguage allows logical parallelism to be expressed in instrument definitions, and uses this information to the extent possible.

The point of contact between the synthesis sublanguage and the score sublanguage is a declaration of "exported" scratch-pad registers from an outermost instrument definition. Multiple instances of instruments can be created, and the externally visible scratch-pad registers associated with each instance can be treated symbolically.

Another aspect of the boundary between score and instrument is the handling of *envelopes*—the piecewise linear functions of time for which many music synthesizers, including the 4C, provide hardware support. 4CED treats envelopes as a special, very limited form of score, capable only of intervening at the "break points" between linear segments to supply a new slope and target value.

The score sublanguage is undoubtedly the most interesting part of 4CED. Each running score acts as an independent event-driven process. Scores can influence each other directly, by triggering events that cause other scores to run, or indirectly, through variables in the global environment.

In more detail, the parallel processing model of 4CED is called *intervention scheduling* [Abbott 1984]. In the intervention scheduling model, each running process is called a *schedule instance*, and is thought of by the programmer as running only at instantaneous, uninterruptible moments, called *interventions*. Each intervention is triggered by an event. Events are caused externally (e.g., when a button or key is pushed), by a programmed delay, or by an explicit command from another schedule instance.

A small expression language is available for interventions, allowing evaluation of arithmetic expressions, assignment to local and global variables, and assignment to scratch-pad registers in the 4C. Data are available in the form of constants, references to array elements, and an operator that reads a potentiometer or other user performance device. Two kinds of arrays are available, one implicitly referenced by the program counter (thus providing data for the "current event"), and another, which functions as a normal (one-dimensional) lookup table. Other operators include control flow (a form of **if–then–else**, and a **goto**), as well as a mechanism to trigger other schedule instances.

The intervention scheduling model is conceptually clean; experience with it indicates that it is appropriate to real-time music synthesis and performance applications. It is also quite general: The ability of schedules to trigger each other allows one to build systems of interacting schedules in ways that extend beyond the simple models of musical voices used in some systems. The strength of 4CED is the great flexibility afforded by its underlying general computational model.

3.3.6 MOXIE

Moxie is a system designed for real-time programming for composition and performance of the NEDCO synthesizer, the same one used for PLAY [Collinge 1980]. It consists of a program preprocessor for the XPL language and a set of XPL subroutines that manage the run-time environment of the NEDCO system. It is an event-oriented system, not too different from 4CED in its overall design. Events may be caused by a performance gesture, a signal from another event, or a delayed signal from the current event. The response to the occurrence of an event is coded in an *action component*, which is written in XPL with Moxie preprocessor directives interspersed. A symbol table is built to associate specific events with actions. An action ordinarily performs some computation or sets some state in the synthesizer. It may also cause the invocation of another action, possibly itself, after a specified delay. A scheduler interprets the resulting time-ordered stream of signals and invokes the appropriate actions.

Piecewise linear functions of time are built into the language, again, to provide support for envelopes. Like the 4C, the NEDCO, synthesizer provides hardware support for linear ramp generation. Thus each break point on a piecewise linear function is associated with an event that loads this hardware with coefficients for the next linear segment.

Since Moxie is a superset of XPL, all the usual facilities of a general programming language are available, in particular, conditional and looping statements. Interevent communication is limited to a few fixed arguments, but these can be used to pass pointers to data structures, as long as the data are in shared static memory. While the processor is idling, a background process can be providing additional state information, for example, by reading a precompiled score, or tracking acoustical inputs. The foreground events and the background process can communicate by causing events recognized as semaphores.

Moxie is a very flexible system. It can be used to write scores that play themselves or that are controlled at a high level by performance inputs, or trivially, to behave like an organ. This shows the power of extending an existing programming language with primitives appropriate to real-time control of a synthesizer. XPL is a compiled language, and so Moxie is too. This is one of its principal conceptual differences from 4CED, which is interpreted. Naturally, Moxie gains in relative performance efficiency by being compiled. What it loses by forcing recompilation for every change to an action component it regains by having a console interpreter, which can dynamically load variables during performance. The console interface can control synthesis, performance, or compositional parameters.

Moxie is under active development. It is being ported to other synthesizers, and a version has been developed for an extended version of the FORTH language. It has been enhanced to include action priorities that allow arbitration between time-critical and non-time-critical events, and some language extensions.

3.3.7 FMX

Recently, a development effort has been undertaken to address the unique audio processing needs of filmmakers. The work is targeted for a large multiprocessor known as the Audio Signal Processor (ASP) built by J. A. Moorer at Lucasfilm [Moorer 1982]. Because of the diverse needs of filmmakers, the project has been able to focus on issues of flexibility and extensibility. Efficiency is also a major issue because of the demands for real-time control over the substantial amounts of signal processing

required for film sound. The main program developed thus far to control the ASP is called FMX.

The signal-processing elements of the ASP are horizontally microcoded using a writable control store (WCS), which can be modified by the controlling computer as easily as the sample data registers. Code to be loaded into the WCS can be time-stamped, facilitating the control of program execution in time. The machine is designed to be easy to microcode. Because of this, implementing a language based on a fixed set of unit generators would be inappropriate. Instead, tools are provided to create arbitrary unit generators. Then an optimization strategy is employed whereby the unit generators to be loaded are stratified in parallel across the available processing resources so as to maximize the utilization of the machine. This approach is in the spirit of designing for efficiency seen in Music V. Unlike Music V, though, the unit generators are written in interleaved microcode. Instruments in FMX are created by loading interleaved microcode corresponding to a collection of unit generators and "plugging" the unit generators together.

The image of plugging things together is familiar to musicians and audio technicians and has the further advantage of being very general. Connecting two "plugs" in FMX has the effect of making two functions of time be equivalent. Thus this model reduces formally to the "functions of time" model of GROOVE, and, to a lesser extent, PLAY. In FMX, the notion is carried somewhat further than in previous systems. In particular, prerecorded sounds are played by "plugging" them into inputs in the synthesizer. Control functions and sound waveform functions are treated equivalently, although the sampling rate for control functions is generally much slower than the audio sampling rate.

FMX is designed to be extensible to account for the generality and extensibility of the ASP. Up to 8 signal-processing elements can be closely coupled in a single ASP, and up to 16 general-purpose controlling computers can be attached to it. The control computers communicate with the ASP directly, and communicate with each other by a dedicated Ethernet.

Current efforts are directed at a more integrated (and ambitious) programming language, which includes the features of FMX and, in addition, a sophisticated score language. The score language will be based on 4CED's intervention scheduling model and implemented more completely.

3.4 High-Level Languages for Computer Music

We said earlier that the trend in music synthesis language design in recent years has been to extend language structure upward from the acoustic level toward higher musical levels. There were some seminal efforts to compute structural elements of music within the context of music synthesis languages [Smith 1976; Smoliar 1971]. Many other efforts have been made at music representation outside of the context of music synthesis languages, as we discussed in the introduction. We are limiting this discussion to those languages that are tightly coupled with synthesis languages or techniques, or that deal with music performance and hence real-time operation.

The languages described in this section are all recent, and try to provide higher level expression for music representation, synthesis, or real-time control than those previously considered. The inputs to these languages are programs and high-level specifications of musical scores. What distinguishes them from general-purpose languages is that they provide many necessary data abstractions and control flow paradigms that are more directly suited to music. These include

- convenient methods of handling the flow of time,
- structural organization of musical materials,
- music data representation, and
- parallelism and hierarchy.

Some of these languages are supersets of other high-level, general-purpose languages. In this case, they are generally written in the languages that they extend. It becomes especially difficult here to differentiate between a program written in a language and a compiler written in a language that implements another language.

In the examples we cite, however, the base language has been extended in some nontrivial way.

Music seems to require at least the same programming language support as any other application. This leads to the conjecture that good general-purpose languages make good music languages [Roads 1982]. The representation of music, however, has additional requirements that are typically not found in general-purpose languages, such as the four items mentioned above. Although this does not invalidate the conjecture, it at least demonstrates that computer music is an excellent testing ground for the extensibility of general-purpose languages. And indeed, it has been the tendency of music language designers—as we saw in the case of MUS10—to extend established languages instead of inventing new ones out of whole cloth.

Two of the languages in this section, Pla and Formes, are written in the programming languages that they extend, SAIL [Reiser 1976] and LISP [Chailloux 1978], respectively. The third language, Arctic, is written in the terms of functional programming [Backus 1978].

Both Pla and Formes utilize an *object-oriented* control flow design, Formes more centrally than Pla. An *object* is a program element that contains both data and *methods* to observe and change those data. Invoking a method is accomplished by sending the object a message. The method can be any action, including forwarding messages, doing I/O, or simply altering the internal state of the object. The principal purpose of the object paradigm is to limit and regularize access to state information in a program. This is achieved by delegating control over various parts of the state to different objects, which then have sole responsibility for changing that state. Other objectives of object-oriented programming include the inheritance of data structures between related objects, and ways of ordering the invocation of methods. Object-oriented programming disciplines reduce the complexity of a program by binding together the data and the associated manipulations of that data. In this way, the usual problems of building complex systems, such as excessive globality of state, and lack of control of side effects, is reduced. Objects then can communicate with simple messages to accomplish complicated actions.

An additional benefit of object-oriented programming for music is that it aids in the conceptualization of parallelism in task allocation. Smalltalk [Krasner 1980] and ZetaLISP [Weinreb and Moon 1981], the two best-known object-oriented languages, lack facilities for handling time and concurrency. Lieberman, however, discusses research being done to provide these facilities to object-oriented languages [Lieberman 1980, 1982].

3.4.1 Pla

Pla is a high-level music programming language that produces as its output "low-level" statements in Music N format [Schottstaedt 1983]. General tools are provided for specifying musical scores, providing a rich environment for data and code representation. The approach taken in Pla is to maximize the ways that a composition can be represented and changed. This is done in several ways, including the following:

- a full complement of general-purpose language constructs;
- extensions to support CPN data, and alternate representations of music data in alphanumeric and functional form, with facilities for translating between them;
- a parallel-processing approach to polyphony, including message passing and *flavors* [Weinreb and Moon 1981];
- facilities for editing all representations;
- an interpretive run-time environment to facilitate interaction;
- libraries of useful functions.

Pla is written in SAIL and reimplements most of the basic data types and operations available in SAIL [Reiser 1976]. Pla is an interpreted language, which maximizes its responsiveness.

Pla provides a core of basic data structures and operations that are directly useful for typical compositional use and provides ways of extending these to other applications. Most of this core is borrowed, in whole or part, from a music composition preprocessor called SCORE [Smith 1976].

We shall digress briefly into a discussion of SCORE for insight into some fundamental points about Pla.

SCORE can best be described as a music data entry system, or as a preprocessor for Music N languages. It was one of the first attempts to develop a text-based musical data representation that derives its metaphores primarily from CPN, as distinct from computational models such as Smoliar [1971]. It allows simple built-in commands to generate complex note lists. SCORE was originally developed as an improvement on the Pass 1 and Pass 2 stages of processing in Music V. SCORE's musical data types and its mode of execution are of particular importance to understanding Pla.

The data types that Pla borrowed from SCORE include the most regular of the dimensions of CPN, including *pitches*, *rhythms*, *tempo*, and *dynamics* in a text format representation. For example, the SCORE pitch symbol *bf5* is taken to mean the B flat in the fifth octave of the piano (932.32 hertz). The rhythmic symbol *Q* means a quarter note, and is translated to a duration of 1 second at a metronome marking of 60 quarter notes per minute. Dynamics are represented in their traditional nomenclature, . . . *pp*, *p*, *mp*, *mf*, *f*, *ff*, . . . , where *pp* is very soft (in Italian, *pianissimo*) and each symbol from left to right is successively greater in amplitude by some amount. *ff* is taken to mean "quite loud."

In SCORE, music is specified by making sequential arrays of these symbols. That is, the first line, or array, might be pitches, which might together constitute the melody; the next array is the rhythms, one for each pitch; then dynamics follows; and last are arrays for parameters dealing with the details of the synthesis technique used by the instrument which will receive the score. In the simple case, a tight correspondence exists between a particular array and one particular parameter field of the resulting Music N note list. Thus elements from an array of pitches might end up as numeric arguments in parameter field 5 (notated **p5**) of a list of note statements. This binding of array to parameter field is stipulated in the syntax of SCORE arrays. Collections of arrays are grouped together to form instruments. SCORE takes fields sequentially from each set of parallel arrays, translating them into note statements. It does this by traversing them in parallel, with an appropriate interpreter for each array type, generating one or more note statements for each index into the parallel arrays. SCORE handles polyphony by collating multiple array sets from simultaneously evaluated instruments in correct time order. Where any array is shorter than the others in an instrument, the interpreter simply wraps around to the beginning, giving rise to the name "cyclic list."

Pla takes this method and expands it in some interesting directions. Pla defines a **voice** as the basic active element of a Pla program. The voice is roughly analogous to the instrument in SCORE. Voices are templates for voice instances. An *instance* of a voice is a data component that contains state information and a program counter for each active instance of a voice. Data for a voice include starting time and ending time for the life span of a voice. A background scheduler notes the times, and when the time arrives for a voice to be run, the scheduler invokes it. When invoked by a scheduler, the voice instance attempts to complete one traversal of its execution tree, which normally results in the generation of one Music N note statement. The voice instance may also make a request to the scheduler to resume its execution at a future time. Any method can be used to generate parameter fields in the note statements, such as generating parameters by algorithm, editing a preexisting note list, traversing functions of time, or reacting to prior output or the behavior of other voices. Voices can directly examine the internal state of other voices, or they can communicate among themselves via messages.

Pla addresses the need for hierarchy in scores by the notion of *sections*, which are collections of voices. A section is identical to a voice, except that it can contain other voices.

The model for voices is really the parallel process run-time facility of SAIL, the same as that used by MUSBOX. Pla, however, capitalizes on the similarity between the parallel-process model and object-oriented programming by including a *flavor* data

type modeled after the one by Weinreb and Moon [1981]. A flavor is a behavior that a voice can exhibit, and is roughly equivalent to a regular object-oriented programming discipline.

An example of message passing in a musical context is as follows. In CPN, musical symbols are often *saturated*, meaning that they stand for a constellation of interpenetrating attributes. Such musical symbols are usually "device independent" in that they will be executed differently by different instruments, but always to the same musical end. For instance, the directive *largo* indicates a complex of performance nuances; it mainly indicates a slow tempo, but also a number of other possible attributes, including a subdued dynamic level, a *legato* style (notes connected with overlaps or slurs), muted timbres, slower vibrato, and longer attacks on notes, to mention a few.

The execution of a *largo* directive will be physically different on different instruments (e.g., pianos cannot do vibrato), but the effect should be the same. A well-designed music composition language should certainly allow for the concise representation of such saturated symbols, singly or in combination. Pla addresses this through its message/flavor system. A composer can send the message *largo* to any voice that contains a method to handle that message. The composer designs the *largo* method to modify the default state of a voice so that it emits note statements in a suitable way. Furthermore, flavors can be "mixed" together, so that it becomes possible to combine such saturated symbols without having to redefine the fundamental function of the voice. For instance, flavors allow us to handle the mixed case *molto largo con espressione* with relative ease, compared with what we would be required to do otherwise. As convenient as messages and objects are, they are only a way to represent behaviors; the trick is to come up with the correct behavior for a *largo* in the first place! The task, however, is certainly facilitated by such techniques.

Pla is a transitionary language. It moved from the primitive origins of SCORE through SAIL, and into the realm of object-oriented languages, picking and choosing as it went. In the process, it has certainly met its goal of providing a rich and flexible compositional environment. On close inspection it is also apparent that Pla has developed a baroque and somewhat eclectic collection of features along the way.

Work continues on Pla, directed toward providing automatic transformations between alternate representations of a score. A composer using Pla can create and interact with arbitrary graphic representations of a score. An example session of using Pla might show a composer developing a multiwindow display, in which one score is represented alternatively as note statements, or continuous functions of time, or as notes on a staff, or as graphical icons.

3.4.2 Formes

Formes is an object-oriented language designed for music composition and synthesis that was developed at the Institute de Recherche et de Coordination Acoustique-Musique in Paris [Cointe and Rodet 1983]. Formes—written in VLISP [Chailloux 1978]—has an advantage over Pla in that its base language already implements object-oriented metaphors. Formes applies a direct object-oriented programming discipline, in which complexity is localized into program objects. The authors of Formes hope that it will facilitate building "models of knowledge" about music. For example, one could provide a model of musical knowledge about the term *largo* by specifying rules for its production. In the previous section we described the term *largo* as a complex symbol denoting numerous intertwined adjustments to some default performance parameters. A program capable of working with such musical knowledge might apply the set of productions associated with *largo* to the music that it was processing whenever it received a directive to do so.

Formes provides a framework for developing these models of knowledge with a built-in method for temporal organization of events, and a built-in genealogical hierarchy of objects. The genealogical tree of objects is organized by giving each object a set of fields that point to the object's parent, left and right siblings, and children.

Each layer of the hierarchy is taken as a musical level, from abstract to concrete. A dynamic execution tree is formed from this genealogical tree on each tick of a clock. The Formes scheduler sequences all active objects on each tick according to a combination of the rules of hierarchy, timing, and certain precedence rules. The tree is rebuilt on each clock tick. Each traversal of the tree corresponds to an update of synthesis parameters, either through direct control of a synthesizer, or via note lists, implementing a synchronous control paradigm.

The goal of Formes is to provide a creative programming environment that contains libraries of predefined musical models, tools for their easy elaboration, and archiving for the successful extensions. The hope is that this will facilitate easy control of musical complexity and provide a method to expand our understanding of musical form.

Of course, many programs can be said to contain models of musical knowledge. The question is always how useful such knowledge is outside the particular context in which it is developed. To address this, the authors of Formes express the following desiderata. Musical models should be as *general* as possible, so as not to be limited by the particulars of any one synthesis technique or sound. The description of musical models should refer to *universal concepts* by being represented in cognitive or psychoacoustic terms whenever possible. Models should be *compatible* with each other so as to combine readily into larger structures. Model building should be through *hierarchy*, allowing complex behaviors to be developed from simpler ones.

3.4.3 Critique of Pla and Formes

Whereas Pla starts with a concern for score representation and deals with control flow secondarily, Formes takes control flow as its departure point and assumes that useful representations will be filled in later. Thus in Pla we see numerous ways to represent and manipulate musical data, but we are presented with a less coherent programming structure, whereas in Formes we see a clear structure, but almost no mention of musical data formats. While it would not be useful to carry the distinction between data structures and control flow too far, it does serve to intuitively characterize the range of approaches.

These languages do not ordinarily operate in real time because their ability to handle complexity requires considerable computing. The composer is expected to work in an iterative fashion. It is easy to imagine using certain features of these languages in real-time control of synthesis systems; for instance, it would be useful to apply the same tools that are used to describe compositional levels to describe levels of performance interpretation. But to do this would require a system that ran in real time.

It seems that there are two streams of music language development: those that primarily address the complexity of music representation and are secondarily concerned with efficiency, such as Pla and Formes, and those that emphasize efficiency in order to address real-time signal processing control, and are less concerned with high-level representation, such as Moxie and 4CED. Ultimately, this gap will have to be closed.

3.4.4 Arctic

Arctic is a very different language from those considered above [Dannenberg 1984]. It implements features of both GROOVE and 4CED within the context of a functional programming language [Backus 1978]. Arctic takes an *applicative* or *declarative* approach, rather than an *procedural* or *imperative* one. Rather than achieving concurrency with multiple threads of control through a set of sequentially executed program modules, as in Pla and Formes, the time domain and concurrency are expressed directly in the language.

Arctic borrows from GROOVE in that the behavior of programs is represented as the traversal of functions of time. New behaviors can result from combining and modifying functions. It resembles 4CED in that events can trigger actions of a program, and like Pla, time is notated explicitly (or implicitly, if so desired). Timing is

established as a consequence of language semantics rather than processing speed or programmed reference to an external clock.

Arctic has two primary elements: *prototypes* and *instances*. A prototype is taken to mean a higher order function whose parameters include an interval of time. A prototype is instantiated when it is given a start time and duration factor, and the result is called an instance. Instances model specific behaviors, and prototypes model parameterized classes of similar behaviors. Prototypes can be either data streams or computed functions; there is no notion, however, of sequential execution or state. Instead, the start time (or event conditions leading to a start time) and duration are stated explicitly in expressions of the language. Addition, multiplication, assignment, and relational operators are available, in addition to functional control constructs as primitives to define prototypes. Variables exist, but they are used only to refer to the same instance from several points in a modular program; they do not store state information.

Two special operators are used to scale the duration factor and offset the start time. If we have a prototype named *Function*, the statement

Function

instantiates it over the implicit time interval [0, 1],

Function @ 5

gives it a time offset of 5 seconds, and

Function @ 5 ~ 2

starts it at time 5, and scales the implicit duration factor by 2. Time can be made to flow retrograde.

Groups of prototypes, called *collections*, can be constructed and given collective start times and duration factors. Individual elements of a collection can have additional time scalings and offsets, which are combined with those inherited from the collection as a whole. Elements can also be executed sequentially by using a sequence operator " | " to replace the normal semicolon (";") concurrency operator.

The Arctic language is an attempt to view time control from a high-level perspective, where the control of time and of concurrency is stated in the language, and the implementation details are hidden. Most other languages that address real-time programming require the programmer to obey a complex set of inferred rules when writing real-time applications, and as a result the tools available to evaluate the correctness of such programs are weak. The elimination of sequence, assignment, and variables that store state means that the evaluation of Arctic programs is not time dependent, insofar as it does not depend on external events.

The problems with Arctic's approach center around the elaboration of the model. A more general metaphor for time is needed to conveniently implement, for instance, tempo variation. Also, the scope of references to instances can be broader than the existence of those instances, leading to situations in which operations can be performed on undefined instances. The expressivity of functions of time is also limited for a wide variety of tasks required in a real-time context. Finally, "burying the implementation details" can go too far and yield an unusably rigid system unless it has as its basis the ability to perform arbitrary computations. The designers are building an interface between Arctic and a more conventional procedural language to alleviate this problem.

4. CONCLUDING REMARKS

Music is a highly structured activity in a rich and complex symbolic domain. Thus the relationship of music and computational formalisms is interesting in both directions. Musical uses of computers can be roughly categorized according to how profoundly the formal infrastructure implicit in computation is integrated into them. Much of the work by computer musicians has focused on tools, for example, tools for analyzing sound, for creating and editing classical music notation, and for encoding classical music notation into a linear form. Although such tools can have an immense practical significance, to the extent that

they simply make existing kinds of musical activity more convenient, they do not reflect any profound influence of computer science on music.

Other kinds of work in computer music are more involved with the deeper implications of computational formalisms. For example, composers have explored the formalization of compositional procedures [Blum 1979; Koenig 1970a, 1970b], music theorists have explored the formalization of musical analysis [Lerdahl and Jackendoff 1983; Lincoln 1970], and performers have investigated the area of performance practice.

A fundamental dividing line exists between programmable and "turnkey" systems, which points up the need for the flexibility of a programming language approach in musical applications. GROOVE presents an interesting case in point. When originally developed, its user interface was a very flexible music data entry system—at least, in the eyes of its implementors. Its best feature was that it included an "escape hatch," where users could substitute their own user interface. It was soon realized, however, that every composer had such different requirements that even the capabilities of very "enlightened" data entry systems were quickly superseded, and a programming language was soon substituted for the data entry system. Much the same dynamics were at work in the development of Pla out of SCORE.

Music languages are on the cutting edge of all kinds of work in computer music. Indeed, both compositional and analytical theories need expressive and well-adapted languages in order to be tested and made effective. Development work that focuses on tools for better defined tasks also benefits enormously from a well-designed and effective notation.

Our belief is that the musical domain provides a rich and interesting arena for the study of computation and formal systems. Musical synthesis is a particularly taxing branch of digital signal processing, and is a natural application for data-flow and stream-oriented computational paradigms. Real-time music synthesis coupled with live performance involves a number of additional difficulties. Event-based computational paradigms have received considerable attention in this area (e.g., 4CED, Moxie). The notion of creating and managing sampled functions of time for control as well as synthesis has also been developed (e.g., GROOVE). Hybrids of these approaches have been developed as well (e.g., PLAY and Arctic).

None of the programs we have surveyed have systematically explored distributed systems, an area whose importance is increasing in computer music as well as computer science. In a musical performance mediated by computers, the nature of the communication between distributed computing sites may be highly symbolic as well as very time dependent (in the sense that a communication must be acted on immediately, as well as in the sense that its significance may depend on a context that is continuously unfolding). This is an intellectually (and computationally) challenging area whose demands are somewhat different from those customarily associated with distributed computing.

Indeed, a number of the problems that computer scientists spend their time thinking about are common to anyone who has some relatively ambitious use for computers. We hope that we have demonstrated that computer musicians are in this category. The widely recognized problem of managing complexity is certainly felt within the computer music community, as can be seen from recent language designs such as Formes and Pla, as well as the immense volume of programs that have been developed at the major computer music research sites.

A number of the current trends in programming language design have excited interest among computer musicians. For example, the object-oriented approach of ZetaLISP or Smalltalk is of interest for several reasons, among which are its potential to facilitate highly modular programs, the ease with which various musical behaviors can be defined idiosyncratically (as in the example of *largo* above), and the fact that it is well adapted to distributed sys-

tems. Pla and Formes incorporate the object-oriented approach to computing to some extent.

Other recent trends in the programming language field, such as the widespread interest in explicitly logic-based computational paradigms (e.g., Prolog), have not yet found their echo among computer musicians. Perhaps as these languages become increasingly available and better understood, and as more serious work is undertaken on higher level representations of music, this will change. An interesting development would be languages that allow a very high-level approach, such as that of Prolog, without sacrificing the possibility of speaking in a low-level idiom, such as would be appropriate for signal processing and synthesis. As we have tried to emphasize, the music application spans an unusually broad number of levels, and this is a part of its challenge and interest.

Up to this point, we view the field of computer music as having been dominated by *technological imperatives*, which have been guiding the development of suitable idioms of musical computation since its inception. We have seen this, for example, in Music V, which models a musical note as an entry point to a data-flow subprogram—a classic application of computer science technique. It would be wrong to say, however, that computer musicians have been merely addressing technological imperatives. Instead, *artistic imperatives* are beginning to shape their work more and more. That is, among musicians who have mastered the lessons of computational formalism, the choice of a particular formalism or design to accomplish some goal is motivated less by technological design considerations and more by the requirements of the artistic product. This is apparent when a composer chooses a highly dimensioned, nonhierarchic representation of the relationships between musical elements instead of using more classical tree structuring, because this method provides more powerful means to analyze and modify the structure. It is evident from languages such as Formes and Pla that the trend in sound synthesis language design is to gracefully extend control over ever-larger structures. We can expect the results of such artistic imperatives to provide further evidence on the nature of the artistic process, with the result that higher level aspects of the musical design process will come into focus, giving light to new goals for the expression of music with computers.

ACKNOWLEDGMENTS

The work of G. Loy was supported by the Systems Development Corporation.

The authors wish to express their gratitude to Curtis Roads and John Strawn for their careful and insightful comments on the draft of this paper. Special thanks are due to Curtis Roads for help in researching the references.

REFERENCES

ABBOTT, C. 1981. The 4CED program. *Comput. Music J. 5*, 1.

ABBOTT, C. 1984. Intervention schedules for real-time programming. *IEEE Trans. Softw. Eng. SE-10*, 1.

BACH, J. S. 1742. *The Six French Suites.* Schmieder numbers 812–817.

BACKUS, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug.), 613–641.

BALZANO, G. J. 1985. What are musical pitch and timbre? *Music Perception 3*, 2. In press.

BARLOW, K. 1980. Bus journey to Parametron. *Die Feedback Papers.* Köln, pp. 21–23.

BENNETT, W. R. 1948. Spectra of quantized signals. *Bell Syst. Tech. J. 27.*

BERG, P. 1979. PILE—A language for sound synthesis. *Comput. Music J. 3*, 1.

BLUM, T. 1979. Herbert Brun: Project Sawdust. *Comput. Music J. 3*, 1.

BUSSONI, F. 1962. Sketch for a new esthetic of music. In *Three Classics in the Aesthetics of Music.* Da Capo Press, New York.

BYRD, D. 1984. Music notation by computer. Doctoral dissertation, Dept. of Computer Science, Indiana University, Bloomington, Ind.

CAGE, J. 1961. *Silence.* MIT Press, Cambridge, Mass.

CAGE, J. 1969. *Notations.* Something Else Press, New York.

CHADABE, J., AND MEYERS, R. 1978. An introduction to the Play program. *Comput. Music J. 2*, 1.

CHAILLOUX, J. 1978. Manuel de référence. Rep. RT 16-78, Université de Paris 8, Vincennes.

CHOMSKY, N. 1965. *Aspects of the Theory of Syntax.* MIT Press, Cambridge, Mass.

CHOWNING, J. 1973. The synthesis of complex audio spectra by means of frequency modulation. *J. Audio Eng. Soc. 21*, 7.

COINTE, P., AND RODET, X. 1983. Formes: A new object-language for managing a hierarchy of events. Internal report, Institut de Recherche et de Coordination Acoustique-Musique, Paris.

COLLINGE, D. 1980. The Moxie user's guide. Internal report, School of Music, Univ. of Victoria, Victoria, B. C., Canada.

DANNENBERG, R. B. 1984. Arctic: A functional language for real-time control. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*. ACM, New York.

DAVID, E. E., JR., MATHEWS, M. V., AND MCDONALD, H. S. 1958. Description and results of experiments with speech using digital computer simulation. In *Proceedings of the 1958 National Electronics Conference*.

ERICKSON, R. 1975. The DARMS project: A status report. *Comput. Humanities 7*, 2.

FRY, C. 1980. YAMIL reference manual. M.I.T. Experimental Music Studio, Massachusetts Institute of Technology, Cambridge, Mass.

FUX, J. J. 1965. *Gradus ad Parnassum*, edited and translated by Alfred Mann with John Edmunds, from the original 18th century edition. Norton, New York.

GREEN, M. 1980. PROD: A grammar-based computer composition program. In *Proceedings of the 1980 International Computer Music Conference*. Computer Music Association, San Francisco, Calif.

GREY, J. M. 1975. An exploration of musical timbre. Rep. STAN-M-2, Dept. of Music, Stanford, Univ., Stanford, Calif.

GROSS, D. 1981. A computer-assisted music course and its implementation. In *Computing in the Humanities*, P. C. Patton and R. A. Holoien, Eds. D. C. Heath, Lexington, Mass.

HILLER, L., AND ISAACSON, L. 1959. *Experimental Music*. McGraw-Hill, New York.

HILLER, L., LEAL, A., AND BAKER, R. A. 1966. Revised MUSICOMP manual. Tech. Rep. 13, School of Music, Experimental Music Studio, University of Illinois, Urbana, Ill.

HOLTZMAN, S. R. 1981. Using generative grammars for music composition. *Comput. Music J. 5*, 1.

HOWE, H. 1975. *Electronic Music Synthesis*. Norton, New York.

KOENIG, G. M. 1970a. Project one. Electronic Music Rep. 2, Institute of Sonology, Utrecht, Netherlands.

KOENIG, G. M. 1970b. Project two. Electronic Music Rep. 3, Institute of Sonology, Utrecht, Netherlands.

KRASNER, G. 1980. Machine tongues VIII: The design of a Smalltalk music system. *Comput. Music J. 4*, 4.

LASKE, O. E. 1980. Toward an explicit cognitive theory of musical listening. *Comput. Music J. 4*, 2.

LERDAHL, F., AND JACKENDOFF, R. 1983. *A Generative Theory of Tonal Music*. MIT Press, Cambridge, Mass.

LIEBERMAN, H. 1980. A preview of Act 1. AI Memo 625, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.

LIEBERMAN, H. 1982. Machine tongues IX: Object-oriented programming. *Comput. Music J. 6*, 3.

LINCOLN, H., ED. 1970. *The Computer and Music*. Cornell Univ. Press, Ithaca, New York.

LOY, G. 1981. Notes on the implementation of MUSBOX: A compiler for the Systems Concepts digital synthesizer. *Comput. Music J. 5*, 1.

MATHEWS, M. V. 1961. An acoustical compiler for music and psychological stimuli. *Bell Syst. Tech. J. 40*.

MATHEWS, M. V. 1963. The digital computer as a musical instrument. *Science 142*.

MATHEWS, M. V., AND MOORE, F. R. 1970. GROOVE—A program to compose, store, and edit functions of time. *Commun. ACM 13*, 12 (Dec.), 715–721.

MATHEWS, M. V., MILLER, J. E., MOORE, F. R., PIERCE, J. R., AND RISSET, J. C. 1969. *The Technology of Computer Music*. MIT Press, Cambridge, Mass.

MCNABB, M. 1981. Dreamsong: The composition. *Comput. Music J. 5*, 4.

MINSKY, M. 1981. Music, mind, and meaning. *Comput. Music J. 5*, 3.

MOORE, F. R. 1982. The Computer Audio Research Laboratory at UCSD. *Comput. Music J. 6*, 1.

MOORER, J. A. 1981. Synthesizers I have known and loved. *Comput. Music J. 5*, 1.

MOORER, J. A. 1982. The Lucasfilm Audio Signal Processor. *Comput. Music J. 6*, 3.

MOORER, J. A., CHAUVEAU, A., ABBOTT, C. EASTTY, P., AND LAWSON, J. 1979. The 4C machine. *Comput. Music J. 3*, 3.

NELSON, G. 1977. MPL—A program library for musical data processing. *Creative Comput.*

PARTCH, H. 1949. *Genesis of a Music*. Univ. of Wisconsin Press, Madison. Reprinted by Da Capo Press, New York, 1974.

PISTON, W. 1978. *Harmony*. Norton, New York.

RABINER, L., AND SCHAFFER, A. 1978. *Digital Processing of Speech Signals*. Prentice-Hall, Englewood Cliffs, N. J.

RAGAZZINI, J. R., AND FRANKLIN, G. F. 1958. *Sampled-Data Control Systems*. McGraw-Hill, New York.

RAMEAU, J.-P. 1965. *Traité de l'harmonie réduite à ses principes naturels*, facsimile of 1722 Paris edition. Broude Brothers, New York.

REISER, J. F. 1976. SAIL. Rep. STAN-CS-76-574, Artificial Intelligence Laboratory, Stanford University, Stanford, Calif.

RISSET, J.-C. 1969. An introductory catalogue of computer synthesized sounds. Unpublished memorandum, Bell Telephone Laboratories, Murray Hill, N. J.

ROADS, C. 1980a. Interview with Marvin Minsky. *Comput. Music J. 4*, 3.

ROADS, C. 1980b. Interview with Max Mathews. *Comput. Music J. 4*, 4.

ROADS, C. 1982. A conversation with James A. Moorer. *Comput. Music J. 6*, 4.

ROADS, C. 1985a. Grammars as representations for music. In *Foundations of Computer Music*, C. Roads, Ed. MIT Press, Boston, Mass.

ROADS, C. 1985b. Research in music and artificial intelligence. *ACM Comput. Surv. 1*, 2 (June), 163–190.

SAMSON, P. 1978. A general-purpose digital synthesizer. *J. Audio Eng. Soc. 28*, 3.

SCHAEFFER, P. 1952. *A la recherche d'une musique concrète.* Éditions du Seuil, Paris.

SCHILLINGER, J. 1941. *The Schillinger System of Musical Composition.* Carl Fischer, New York. Reprinted by Da Capo Press, New York, 1978.

SCHOENBERG, A. 1950. *Style and Idea.* Philosophical Library, New York.

SCHOTTSTAEDT, B. 1983. Pla: A composer's idea of a language. *Comput. Music J. 7*, 1.

SHEPARD, R. N. 1964. Circularity in judgements of relative pitch. *J. Acoust. Soc. Am. 36.*

SMITH, L. 1976. SCORE—A musician's approach to computer music. *J. Audio Eng. Soc. 20*, 1.

SMOLIAR, S. W. 1971. A parallel processing model of musical structures. Rep. AI TR-242, Dept. of Computing and Information Science, Massachusetts Institute of Technology, Cambridge, Mass.

SUNDBERG, J., ASKENFELT, A., AND FRYDEN, L. 1983. Musical performance: A synthesis-by-rule approach. *Comput. Music J. 7*, 1.

TENNEY, J., AND POLANSKY, L. 1980. Temporal gestalt perception in music. *J. Music Theor. 24*, 2.

TOVAR, J., AND SMITH, L. 1976. MUS10 manual. Internal user's manual, Center for Computer Research in Music and Acoustics, Stanford Univ., Stanford, Calif.

TRUAX, B. 1977. The POD system of interactive composition programs. *Comput. Music J. 1*, 3.

VARÈSE, E. 1971. The liberation of sound. In *Perspectives on American Composers*, B. Boritz and E. Cone, Eds. Norton, New York.

VERCOE, B. 1979. Music 11 reference manual. Internal user's manual, Experimental Music Studio, Massachusetts Institute of Technology, Cambridge, Mass.

WEINREB, D., AND MOON, D. 1981. Lisp machine manual, 4th ed. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.

WENKER, J. 1972. MUSTRAN II—An extended music translator. *Comput. Humanities 7*, 2.

WESSEL, D. 1979. Timbre space as a musical control structure. *Comput. Music J. 3*, 2.

XENAKIS, I. 1971. *Formalized Music.* Indiana Univ. Press, Bloomington, Ind.

Received March 1984; final revision accepted August 1985.